NVRAM and Burst Buffers

CSE 5194.01
Spring ‘20

Bharath Ramesh
The Ohio State University
E-mail: ramesh.113@osu.edu
Outline

• Introduction
• Optimizing Checkpoints Using NVM as Virtual Memory
• Mojim
• Boldio
What is an NVM?

- Short for non-volatile memory
- Stores data even when the power is switched off
- Also referred to as persistent storage
- Examples: Flash drives, Hard disks, PCM etc
Optimizing Checkpoints Using NVM as Virtual Memory

S. Kannan, A. Gavrilovska, K. Schwan and D. Milojicic

In 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, Boston, MA, 2013, pp. 29-40.


Note: Images and some text used in the S-Caffe section are taken from the authors of the paper
Checkpoint/Restart

• Failure rate of applications is expected to increase
  – Need Fault tolerant mechanisms with low impact on applications

• A checkpoint is a snapshot of application state stored in persistent devices

• Checkpoint Restart involves using saved snapshots and restarting execution from that point on failure
Challenges in doing Checkpoint/Restart

- Increasing problem sizes and failure rates
  - Leads to increase in checkpoint sizes/processor
  - Rate of increase greater than available I/O bandwidth
- Scalability cannot just be obtained through parallel file systems
  - Limited I/O bandwidth
  - Contention at I/O subsystem level
More information on Checkpoints

• 2 Types of checkpoint mechanisms
  – Application-initiated (preferred for HPC) - Store only data structures identified by application developers
  – Transparent – Save an entire processes' address space

• Multi-level checkpoints - Temporarily store checkpoint data at multiple locations at remote neighbors (i.e., peers)
  – Improves application reliability
Contributions of paper

• Using NVMs for multi-level checkpoints
  – Leveraging NVM features that include byte-addressability, hardware support for memory management, and caching

• Novel checkpoint ‘pre-copy’
  – Reduces the total data volume to be moved at checkpoint time

• Remote memory access interface to NVM reducing network contention

• Benefits evaluated with 3 well known HPC applications
Using NVM as Virtual memory

• Use NVM as virtual memory (instead of a fast disk). It has the advantages listed below.
  – Ability to exploit VM paging and protection
  – Hiding high write latencies using processor cache
  – Byte addressability and hence avoiding serialization
  – Ability to use NVM not only for storage, but also as heap for processing.
  – Current file systems are not optimized for NVMs and require redesign.
Using NVM as Virtual memory

- Maintain per-process NVM container
  - Contains persistent data
- Process address space is extended to NVMs
- Eliminates file system serialization overheads
- Enables memory protection mechanisms for pre-copying
Shadow buffering

- Handle heap allocation for checkpoint data
  - Create a DRAM and a “shadow” NVM chunk for data
  - Reads can happen directly on NVM
  - Any writes by application will copy data to DRAM and then perform the operation

- Overcomes slow NVM writes
Dealing with limited NVM bandwidth using pre-copy

- Copy bandwidth reduces with increase in cores
- Authors propose “chunk-level” pre-copy mechanisms
  - Moves data asynchronously to NVM before checkpoint is started.
  - Enables overlap of computation and checkpoint
Chunk-based pre-copy (CPC)

- Use chunk level protection
  - pre-copy and write protect chunks
  - After pre-copy step, if chunks are modified, mark them as “dirty”
  - Dirty chunks are pre-copied again

- During a local checkpoint, copy only remaining dirty chunks
  - Reduces peak bandwidth utilization

C, L, R denotes compute, local and remote checkpoint step respectively.

Figure a. shows basic sequential local and non-blocking remote checkpoint.

Figure b. shows a pre-copy local checkpoint with compute and local checkpoint steps partially overlapped.
Disadvantages of Chunk-based pre-copy

- Applications can randomly modify chunks
  - Leads to overhead due to repeated pre-copy from DRAM to NVM

- pre-copy increases the protection fault overhead and the amount of data movement across the DRAM-NVM boundary

- Solution: Delayed Chunk Pre-Copy (DCPC)
  - waits for the first checkpoint step to complete and finds the approximate interval (checkpoint time - compute start time)
  - determine a pre-copy starting time called pre-copy threshold using:
    \[ T_c(s) = \frac{D}{NVMBW_{\text{core}}} \]
    \[ T_p(s) = I - T_c \]
    where \( T_c \) is Checkpoint time, \( T_p \) is pre-copy threshold time, \( D \) is checkpoint data size (MB), \( I \) is the checkpoint interval, and \( NVMBW_{\text{core}} \) is Effective NVM BW/core.
Delayed Pre-Copy with Prediction (DCPCP)

- In some cases chunks can be modified during the completion of a compute phase
  - Called “hot chunks”
- Prediction captures frequency of chunk modification
- Chunk not copied to NVM until the modification count is equal to or greater than the value in the prediction table

DCPCP is used in remote checkpoints as well

Figure 6. PreCopy with Prediction
Implementation APIs used

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>genid(varname)</td>
<td>generate id from varname</td>
</tr>
<tr>
<td>nvalloc(id, size, pflg)</td>
<td>allocate NVM memory. pflg -whether data should be persistent</td>
</tr>
<tr>
<td>nv2dalloc(..dim1, dim2..)</td>
<td>2D Fortran allocation wrapper</td>
</tr>
<tr>
<td>nvattach(id, src, size)</td>
<td>create shadow NVM copy for existing DRAM memory</td>
</tr>
<tr>
<td>nvreallocate(id, src, size)</td>
<td>grow memory</td>
</tr>
<tr>
<td>nvcchkptall()</td>
<td>checkpoint all persistent chunks</td>
</tr>
<tr>
<td>nvcchkptid(id)</td>
<td>checkpoint specific chunks/variables</td>
</tr>
</tbody>
</table>

Table III

NVM Checkpoint Interfaces
Results for local checkpoints with pre-copy
Results for remote checkpoints with pre-copy
Results for remote checkpoints with pre-copy
Summary

- NVM as virtual memory has distinct advantages
- Use of pre-copy helps deal with NVM bandwidth limitations
- Remote checkpointing peak bandwidth is reduced with pre-copy
- Focus is on improving efficiency and overlap of checkpointing
Mojim: A Reliable and Highly-Available Non-Volatile Memory System

Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson.

In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’15)

https://dl.acm.org/doi/10.1145/2786763.2694370

Note: Images and some text used in the S-Caffe section are taken from the authors of the paper
What is Mojim?

• A system that provides the reliability and availability for large scale storage systems while preserving the performance of NVMM.

• Uses a two-tier architecture.
  – primary tier contains a mirrored pair of nodes
  – secondary tier contains one or more secondary backup nodes with weakly consistent copies of data.

• Uses highly optimized replication protocols, software, and networking stacks to minimize replication costs and expose as much of NVMM’s performance as possible
Reliability vs Availability

- Reliability – Measures estimated time a system can perform correctly (measures failure rate). Example: MTBF
- Availability – Measures Fraction of total time the system performs correctly
- Subtle difference
  - Reliability = (total time – time in failures) / total failures
  - Availability = (total time – time in failures) / total time
Mojim interface

• Consists of interfaces to primary, mirror and backup nodes
  – Primary nodes supports read/write to replicated data
  – mirror and backup nodes support read only

• A system configuration file specifies a set of Mojim regions on the primary node to be replicated, along with a mirror node and a list of backup nodes where the replicas should reside

• applications can create files in the Mojim-backed file system and map them into their address space using mmap
Mojim interface

- Provides sync points that allow applications to control when and what updates in the data area propagate to the replicas
  - One uses msync, a system call that flushes changes made in-core to persistent storage to guarantee that a copy exists there.
  - Replicates replicas atomically, unlike the regular msync that does not provide such guarantees
- Proposed “gmsync” adds the ability to specify multiple memory regions for the sync point to replicate, allowing for more flexibility than msync.
Mojim Usage Example

```c
int fd = open("/mnt/mmapfile", O_CREAT|O_RDWR);   // open a file in mounted Mojim region
void *base = mmap(NULL, 40960, PROT_WRITE,
          MAP_SHARED, fd, 0);                      // mmap a 40KB area in the file

unsigned long *access_count_p = base;
unsigned long *log_size_p = base + sizeof(unsigned long);  // size of the log
int *log = base + 2*sizeof(unsigned long);

*access_count_p = *access_count_p + 1;
msync(access_count_p, sizeof(unsigned long), MS_SYNC);     // memory load and store

int beautiful_num = 24;
unsigned long curr_log_pos = *log_size_p;
log[curr_log_pos] = beautiful_num;
*log_size_p = *log_size_p + 1;
struct msync_input { void *address; int length; };
struct msync_input input[2];
input[0].address = &log[curr_log_pos];
inpu[0].length = sizeof(int);
inpu[1].address = log_size_p;
inpu[1].length = sizeof(unsigned long);
gmsync(input, 2, MS_MOJIM); // call gmsync to commit the log append
```
Mojim architecture

- Two-tiers – primary, mirror and secondary
- Primary node replicates data to its mirror node at each sync point
- Can tolerate N – 1 failures
- Primary and mirror node connected by Infiniband link
- Mirror node replicates data to secondary asynchronously
Mojim replication example
Mojim – Modes and replication protocols

- **S-unreplicated** - A single machine must flush an msync’d memory region from the processor’s caches to ensure data persistence.

- **M-sync** - when an application calls msync or gmsync, the following steps happen:
  1. Push data from primary node to mirror node using RDMA
  2. Write data in mirror node log
  3. Primary node waits for mirror node ack and returns from call.
  4. Mirror node checkpoints to apply log contents to data area.

- **M-syncflush** – Flushes data from primary node caches, to ensure cache coherence. However, RDMA transfers are cache coherent as well, so not necessary.
Mojim – Modes and replication protocols

- **M-async** – Similar to M-sync but does not wait for ack from mirror node. Provides weaker consistency and mandates flushing of cache to maintain persistence.
- **M-syncdisk** - Similar to M-sync, but stores mirror node data area in a hard disk or SSD for monetary savings.
- **M-syncsec** - M-syncdisk with a secondary tier to provide reliability and availability.
- **M-syncseceth** – Similar to M-syncsec, but uses ethernet between primary and mirror nodes to save cost.
### Mojim – Modes and replication protocols

<table>
<thead>
<tr>
<th>Scheme</th>
<th>R</th>
<th>A</th>
<th>C</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-unreplicated</td>
<td>0</td>
<td>Worst</td>
<td>N/A</td>
<td>Low</td>
</tr>
<tr>
<td>M-async</td>
<td>1</td>
<td>Good</td>
<td>Weak</td>
<td>Fair</td>
</tr>
<tr>
<td>M-sync</td>
<td>1</td>
<td>Good</td>
<td>Strong</td>
<td>Fair</td>
</tr>
<tr>
<td>M-syncdisk</td>
<td>1</td>
<td>OK</td>
<td>Strong</td>
<td>Low</td>
</tr>
<tr>
<td>M-syncsec</td>
<td>$N-1$</td>
<td>Best</td>
<td>Strong+Weak</td>
<td>High</td>
</tr>
<tr>
<td>M-syncseceth</td>
<td>$N-1$</td>
<td>Good</td>
<td>Strong+Weak</td>
<td>Fair</td>
</tr>
<tr>
<td>E-writeall</td>
<td>$N-1$</td>
<td>Best</td>
<td>Strong</td>
<td>High</td>
</tr>
<tr>
<td>E-chain</td>
<td>$N-1$</td>
<td>Best</td>
<td>Strong</td>
<td>High</td>
</tr>
<tr>
<td>E-broadcast</td>
<td>$N-1$</td>
<td>Best</td>
<td>Strong</td>
<td>High</td>
</tr>
</tbody>
</table>

- **R** – Number of failures that can be tolerated
- **A** – atomicity
- **C** – Consistency

The last three rows compare with existing replication schemes.
Mojim – Results

Figure 4. msync Latency with DRAM and NVMM. The average 4 KB msync latency with PMEP’s DRAM and NVMM modes.

Figure 5. msync Throughput with DRAM and NVMM. The 4 KB msync bandwidth with PMEP’s DRAM and NVMM modes.

Figure 6. msync Latency with DRAM-based machines. The average 4 KB msync latency with S-unreplicated and Mojim two-tier architecture.

Figure 7. msync Throughput with DRAM-based machines. The 4 KB msync throughput with S-unreplicated and Mojim two-tier architecture.
Mojim – Results

Figure 8. Average *msync* Latency with Different *msync* Sizes on Emulated NVMM. The average latency of *msync* operation on NVMM with request sizes from 8 bytes to 12 KB.

Figure 9. Throughput with Different Application Threads on Emulated NVMM. The *msync* throughput with 1 to 12 threads performing *msync*.
Mojim – Results

**Figure 10.** Filebench Throughput with Emulated NVMM. The throughput of three Filebench workloads with single machine and no replication, the M-asyn mode, and the M-sync mode.

**Figure 11.** Google Hash Table Average Latency with Emulated NVMM. The average latency of sequentially and randomly inserting key-value pairs to the Google dense hash table.

**Figure 12.** Insert Avg Latency. Average latency of inserting key-value pairs on emulated NVMM.

**Figure 13.** Insert Throughput. Throughput of inserting key-value pairs on emulated NVMM.

**Figure 14.** YCSB Average Latency. Average latency of YCSB workloads on emulated NVMM.
Summary

• Mojim can provide replication with small cost
• Can even outperform un-replicated systems
• Paves the way for deploying NVMM in data centers that wish to take advantage of NVMM’s enhanced performance but require strong guarantees about data safety.
Boldio: A hybrid and resilient burst-buffer over lustre for accelerating big data I/O

D. Shankar, X. Lu and D. K. Panda


Note: Images and some text used in the S-Caffe section are taken from the authors of the paper
What is a burst buffer?

- Enables data to be temporarily buffered via high-performance storage layer deployed in a separate set of compute/storage/large-memory nodes before persisting to file system.
Contributions

• High-performance and fault-tolerant burst-buffer client design
• Pipelined RDMA-based Memcached server with hybrid memory support for efficient use of available high-speed SSDs for extended cache capacities
• Resilient persistence manager within the Memcached server for asynchronously persisting data to Lustre, conforming to the shared-nothing architecture by leveraging replicated key/value pairs
• Advanced pre-fetching of cached I/O from the burst buffer layer for improved read performance
• Co-designing a light-weight Hadoop File System abstraction that interfaces the MapReduce/Spark applications with the proposed burst-buffer system over Lustre, in a transparent manner
Burst buffer system (1c) big data I/O potential

- Non-blocking semantics
  - client-side waits and server-side I/O can be overlapped with other data access operations or computations

- Peak throughput offered by a remote RDMA-based Memcached server (2.4–3.1 GB/s) is comparable to that of high-speed SSDs (2.32–3.2 GB/s). On the other hand, local HDDs achieve a peak throughput of 323 MB/s and global parallel storage i.e., Lustre peaks at 955 MB/s.
Boldio architecture

- 3 major components
  - Boldio Burst-Buffer Client (BBC)
  - Boldio Burst-Buffer Server (BBS)
  - Boldio Burst-Buffer Persistence Manager (BBP)
- Client-initiated replication
- File system abstractions
- Uses RDMA semantics
- Shared-Nothing Memcached
  - Each BBS operates independently
High performance Boldio Client Side Design

- **Writes**: Request from FREE_POOL, send to REQ_QUEUE and buffer returned to FREE_POOL
- **Reads**: Read data + offset placed in REQ_QUEUE and application is notified once data is available
- **Enables Non-blocking I/O semantics** (memcached_iget/memcached_iset), (memcached_wait/memcached_test)
High performance Boldio Client Side Design

• Data distribution schemes
  – 1F-MS: One file multiple servers. Scattered using consistent hashing.
    server key ← consistent hash(key) + replica id
  – 1F-1S: One file one server. Files are co-located with metadata keys.
    server key ← consistent hash(prefix(key)) + replica id
Resilient Boldio Server with Persistence Manager

- Decouple communication phase from the data processing phase
- The communicator receives requests into FREE_POOL and queues it at the memory manager via RECV_QUEUE
- The memory manager asynchronously processes these Set/Get requests and notifies the communicator via SEND_QUEUE
Resilient Boldio Server with Persistence Manager

- Coordinate asynchronous file flush via Lustre by writing ‘commit files’ for each Hadoop I/O file that is buffered into the Lustre output directory
  - Commit files written atomically using POSIX file atomics. Represent last successful write
- BBP composed of I/O scheduler thread + pool of I/O workers backed by dirty queues
  - I/O scheduler dynamically scans k/v pairs to identify chunks that need to be persisted and schedules flush by queueing into dirty queues
  - The I/O worker basically computes file name and offset from the key and writes the value at the designated offset for every key/value pair in its dirty queue]
  - On subsequent scans, I/O scheduler updates commit file and reschedules request in case of write failure.
Resilient Boldio Server with Persistence Manager

- Data resilience is achieved using enhanced BBP designs
  - Replica-Aware Timers for Key/Value Pairs: \( \text{Timer(key)} \leftarrow \text{System.getCurrentTime()} + \text{GLOBAL\_TIMEOUT} \times \text{RepID(key)} \). Basically represents probable time to flush
  - Timeout-based I/O Request Scheduling: I/O scheduler schedules only k/v pairs that expire the earliest (when scheduling to the dirty queues)
Boldio micro benchmark results

Figure 9. Evaluation with Memcached-based Micro-benchmarks
Boldio cluster failures results

Figure 13. Handling Dynamic Cluster Failures Boldio Vs. Alluxio-Remote
Summary

• The paper involves Co-design of 3 components to enable an efficient burst-buffer design for Big Data I/O
• Optimal I/O request overlap using non-blocking queues and pipelines.
• Shared-nothing persistence protocol to ensure resilience
• Provides 3x – 7x improvements over direct lustre and 2-10x over Alluxio in cases where reliance on local memory/storage is not feasible